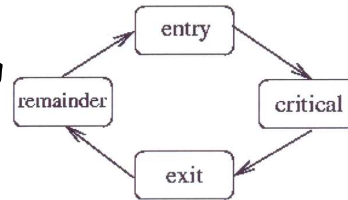
Section 2
Mutual Exclusion

The Mutual Exclusion Problem

- The problem concerns a group of processors which occasionally need access to some resource that cannot be used simultaneously by more than a single processor.
- Examples of what the resource may be:
  - The printer or any other output device
  - A record of a shared database or a shared data structure, etc.
- Each processor may need to execute a code segment called critical section, such that at any time:
  - at most one processor is in the critical section
  - If one or more processors try to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever.

The Mutual Exclusion Problem



Entry (Trying) Section: the code executed in preparation for entering the critical section

Critical Section: the code to be protected from concurrent execution

Exit Section: the code executed upon leaving the critical section

Remainder Section: the rest of the code

Each process cycles through these sections in the order: remainder, entry, critical, exit.

The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.

Mutual Exclusion Algorithms

- An algorithm solves the mutual exclusion problem if the following hold:
  - **Mutual Exclusion**
    In every configuration of every execution, at most one process is in the critical section.
  - **No Deadlock**
    In every execution, if some process is in the entry section in some configuration, then there is a later configuration in which some process is in the critical section.
- Stronger Progress Property
  - **No lockout (starvation-free)**
    In every execution, if some processor is in the entry section in a configuration, then there is a later configuration in which that same processor is in the critical section.

Mutual Exclusion Algorithms

Assumptions

- Any variable that is accessed in the entry or the exit section of the algorithm cannot be accessed in any of the other two sections.
- No process stays in the critical section forever.
- The exit section consists of a finite number of steps.

ME Algorithms that use RW Registers

Algorithms

- Algorithms for two processes
- An algorithm that guarantees mutual exclusion and no lockout but uses O(n) registers of unbounded size.
- An algorithm that guarantees mutual exclusion and no lockout using O(n) registers of bounded size.

Lower Bounds

- Any algorithm that provides mutual exclusion, even with the weak property of no deadlock, must use n distinct RW registers, regardless of the size of these registers.

Proposed solution I

Process p_0	Process p_1
while (true} {	while (true} {
while (turn = 1) noop; //entry	while (turn = 0) noop; //entry
critical section	*critical section*
turn = 1 // exit	turn = 0 // exit
remainder section	remainder section
}	}

turn

0/1

✓mutual exclusion
X deadlock-freedom

Does it work?

Proposed Solution II

Process p_0	Process p_1
while (TRUE) {	while (TRUE) {
flag[0] = true	flag[1] = true
while (flag[1]) {skip}	while (flag[0]) {skip}
critical section	*critical section*
flag[0] = false	flag[1] = false
remainder section	remainder section
}	}

flag

0	false
1	false

✓mutual exclusion
X deadlock-freedom

Does it work?

Proposed solution III

Process p_0	Process p_1
while (TRUE) {	while (TRUE) {
while (flag[1]) {skip}	while (flag[0]) {skip}
flag[0] = true	flag[1] = true
critical section	*critical section*
flag[0] = false	flag[1] = false
remainder section	remainder section
}	}

flag

	flag
0	false
1	false

x mutual exclusion
✓Deadlock-freedom

Does it work?

Peterson's algorithm

Process p_0	Process p_1
While (TRUE) {	While (TRUE) {
flag[0] = true	flag[1] = true
turn = 1	turn = 0
while (flag[1] and turn == 1) {skip}	while (flag[0] and turn == 0) {skip}
critical section	*critical section*
flag[0] = false	flag[1] = false
remainder section	remainder section
}	}

flag

	flag
0	false
1	false
turn	0/1

ME Algorithm using Single-Writer binary RW registers

want[0]: SW register written by p_0 and read by p_1 with initial value 0; it is set to 1 to identify that process p_0 wants to enter the critical section

want[1]: symmetric to want[0]

Process p0

```
while (TRUE) {


3.      want[0] = 1;


6.      wait until (want[1] == 0);
        critical section;
8.      want[0] = 0;
        remainder section;
}
```

Process p1

```
while (TRUE) {
1.      want[1] = 0;
2.      wait until (want[0] == 0);
3.      want[1] = 1;
4.      if (want[0] == 1) then
5.              goto line 1

        critical section;
8.      want[1] = 0;
        remainder section;
}
```

Is this correct?

How can we prove it?

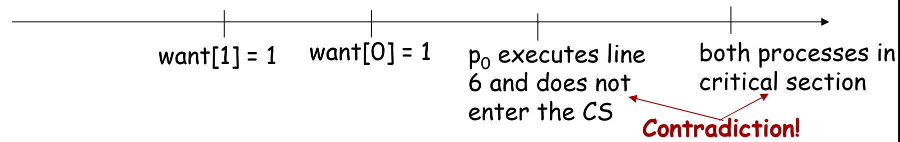
Proving Correctness

Theorem

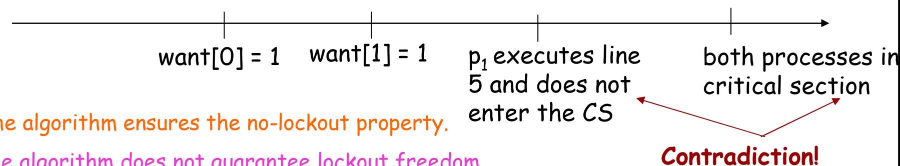
- The algorithm ensures the mutual exclusion property.

Sketch of Proof

- Assume, by contradiction, that at some configuration C, both processes are in the critical section
  $\Rightarrow$ want[0] = want[1] = 1.
- Case 1: Last write of p_0 to want[0] follows the **last** write of p_1 to want[1].



- Case 2: Last write of p_1 to want[1] follows the **last** write of p_0 to want[0].



☺ The algorithm ensures the no-lockout property.

☹ The algorithm does not guarantee lockout freedom.

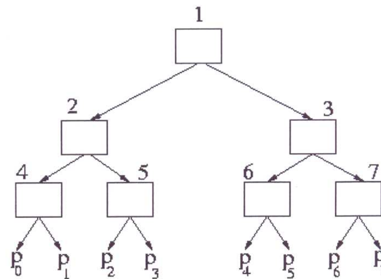
ME Algorithm using Single-Writer binary RW registers – Symmetric Version

Code for process p_i, i = 0, 1

```
while (TRUE) {
1:      want[i] = 0;
2:      wait until ((want[1-i] == 0) OR (priority == i));
3:      want[i] = 1;
4:      if (priority == 1-i) then {
5:              if  (want[1-i] == 1) then
                        goto line 1; }
6:      else wait until (want[1-i] == 0);
        critical section;
7:      priority = 1-i;
8:      want[i] = 0;
        remainder section;
}
```


ME Algorithms for many processes

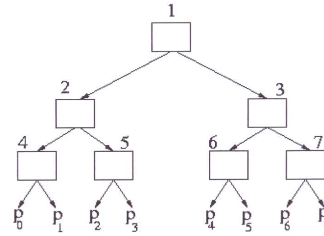
- Processes compete pairwise, using a two-process algorithm.
- The pairwise competitions are arranged in a complete binary tree.
- The tree is called the tournament tree.



- Each process begins at a specific leaf of the tree
- At each level, the winner moves up to the next higher level, and competes with the winner of the competition on the other side.
- The process on the left side plays the role of p_0, while the process on the right side plays the role of p_1.
- The process that wins at the root enters the critical section.

ME Algorithms for many processes

```
procedure Node(v: integer, side: 0..1) {
    1: want^v[side] = 0;
    2: wait until ((want^v[1-side] == 0)
                  OR (priority^v == side));
    3: want^v[side] = 1;
    4: if (priority^v == 1-side) then {
    5:       if (want^v[1-side] == 1) then
                  goto line 1; }
    6: else wait until (want^v[1-side] == 0);
    8: if (v == 1) then
    9:    critical section;
    10: else Node(⌊v/2⌋, v%2)
    11: priority^v = 1-side;
    12: want^v[side] = 0;
}
```



- Tree nodes are numbered. The number of the root is 1. The number of the left child node of a node v is 2v, and the number of the right child of v is 2v+1.
- $want^v[0]$, $want^v[1]$, $priority^v$: variables associated to node v for the instance of 2-ME that is executed at this node.
- Process p_i begins by calling Node($2k+\lfloor i/2 \rfloor$, i % 2), where $k = \lceil \log n \rceil - 1$.

The Bakery Algorithm

for each i, $0 \le i \le n-1$:
Choosing[i]: it has the value TRUE as long as p_i is choosing a number
Number[i]: the number chosen by p_i

Code for process p_i, $0 \le i \le n-1$

Initially, Number[i] = 0, και
Choosing[i] = FALSE, for each i, $0 \le i \le n-1$

```
Choosing[i] = TRUE;
Number[i] = max{Number[0], …, Number[n-1]}+1;
Choosing[i] = FALSE;
for j = 0 to n-1, j ≠ i, do
    wait until Choosing[j] == FALSE;
    wait until ((Number[j] == 0) OR ((Number[j], j) > (Number[i], i)));
critical section;
Number[i] = 0;
remainder section;
```

The Bakery Algorithm

Lemma

- In every configuration C of any execution a, if p_i is in the critical section, and for some k ≠ i, Number[k] ≠ 0, then (Number[k],k) > (Number[i],i).

Sketch of Proof

- Number[i] > 0
- p_i has finished the execution of the for loop (in particular, the 2nd wait statement for j = k).
- ***Case 1***: p_i read that Number[k] == 0
- ***Case 2***: p_i read (Number[k],k) > (Number[i],i)

Theorem

- The Bakery algorithm ensures the mutual exclusion property.

The Bakery Algorithm

Theorem

- The Bakery algorithm provides no lockout.

Sketch of proof

- Assume, by the way of contradiction, that there is a starved process.
- All processes wishing to enter the critical section eventually finish choosing a number.
- Let p_j be the process with the smallest (Number[j],j) that is starved.
- All processes entering the critical section after p_j has chosen its number will choose greater numbers, and therefore will not enter the critical section before p_j.
- Each process p_k with Number[k] < Number[j] will enter the critical section and exit it.
- Then, p_j will pass all tests in the for loop and enter the critical section.

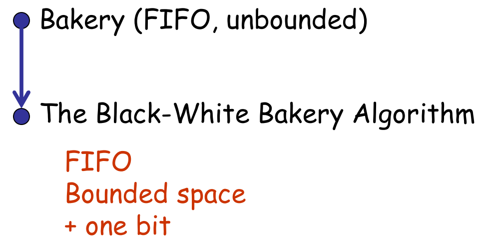
Space Complexity

- The Bakery Algorithm uses 2n single-writer RW registers. The n Choosing[j] variables are binary, while the n Number[j] variables are unbounded, $0 \leq j \leq n-1$.

Bakery Algorithm versus

Properties of the Bakery Algorithm

- The Bakery Algorithm satisfies mutual exclusion & FIFO.
- The size of number[i] is unbounded.



FIFO
Bounded space
+ one bit

The Black-White Bakery Algorithm

```
choosing[i] = true;
mycolor[i] = color;
number[i] = 1 + max{number[j] | (1 ≤ j ≤ n) ∧ (mycolor[j] = mycolor[i])};
choosing[i] = false;
for j = 0 to n {
    await (choosing[j] == false);
    if (mycolor[j] == mycolor[i])
    then await (number[j] == 0) ∨ (number[j],j) ≥ (number[i],i) ∨
                (mycolor[j] ≠ mycolor[i]);
    else await  (number[j] == 0) ∨ (mycolor[i] ≠ color) ∨
                (mycolor[j] == mycolor[i]);
}
critical section;
if (mycolor[i] == black) then color = white;
else color = black;
number[i] = 0;
```

The One-Bit Algorithm

Code of process p_i , $i \in \{1 ,..., n\}$

```
repeat {
    b[i] = true; j = 1;
        while (b[i] == true) and (j < i) {
             if (b[j] == true) {
                         b[i] = false; await (b[j] ==false);
                }
             j = j+1
        }
}
until (b[i] == true);
for (j = i+1 to n)
      await (b[j] == false);
critical section
b[i] = false;
```

Properties of the One-Bit Algorithm

- Satisfies mutual exclusion and deadlock-freedom
- Starvation is possible
- It is not symmetric
- It uses only n shared bits and hence it is space optimal

Bibliography

These slides are based on material that appears in the following books:

- H. Attiya & J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, Morgan Kaufmann, 1998 (Chapter 4)
- G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Pearson / Prentice Hall, 2006 (Chapter 2)
- N. Lynch, Distributed Algorithms, Morgan Kaufmann, 1996 (Chapter 10).